To appear in the *Proceedings of the 26th International Symposium on Computer Architecture (ISCA 26)*, 1999.

Is SC + ILP = RC?

Chris Gniady, Babak Falsafi, and T. N. Vijaykumar
School of Electrical & Computer Engineering
Purdue University
1285 EE Building
West Lafayette, IN 47907
impetus@ecn.purdue.edu, http://www.ece.purdue.edu/~impetus

Abstract

Sequential consistency (SC) is the simplest programming interface for shared-memory systems but imposes program order among all memory operations, possibly precluding high performance implementations. Release consistency (RC), however, enables the highest performance implementations but puts the burden on the programmer to specify which memory operations need to be atomic and in program order. This paper shows, for the first time, that SC implementations can perform as well as RC implementations if the hardware provides enough support for speculation. Both SC and RC implementations rely on reordering and overlapping memory operations for high performance. To enforce order when necessary, an RC implementation uses software guarantees, whereas an SC implementation relies on hardware speculation. Our SC implementation, called SC++, closes the performance gap because: (1) the hardware allows not just loads, as some current SC implementations do, but also stores to bypass each other speculatively to hide remote latencies, (2) the hardware provides large speculative state for not just processor, as previously proposed, but also memory to allow out-of-order memory operations, (3) the support for hardware speculation does not add excessive overheads to processor pipeline critical paths, and (4) well-behaved applications incur infrequent rollbacks of speculative execution. Using simulation, we show that SC++ achieves an RC implementation's performance in all the six applications we studied.

1 Introduction

Multiprocessors are becoming widely available in all sectors of the computing market from desktops to high-end servers. To simplify programming multiprocessors, many vendors implement shared memory as the primary system-level programming abstraction. To achieve high performance, the shared-memory abstraction is typically implemented in hardware. Shared-memory systems come with a variety of programming interfaces—also known as memory consistency models—offering a trade-off between programming simplicity and high performance.

Sequential consistency (SC) is the simplest and most intuitive programming interface [9]. An SC-compliant memory system appears to execute memory operations one at a time in program order. SC's simple memory behavior is what programmers often expect from a shared-memory multiprocessor because of its similarity to the familiar uniprocessor memory system. Traditionally, SC is believed to preclude high performance because conventional SC implementations would conservatively impose order among all memory operations to satisfy the requirements of the model. Such implementations would be prohibitively slow especially in distributed shared memory (DSM) where remote memory accesses can take several times longer than local memory accesses.

To mitigate performance impact of long latency operations in shared memory and to realize the raw performance of the hardware, researchers and system designers have invented several relaxed memory models [3,2,6]. Relaxed memory models significantly improve performance over conventional SC implementations by requiring only some memory operations to perform in program order. By otherwise overlapping some or all other memory operations, relaxed models hide much of the memory operations' long latencies. Relaxed models, however, complicate the programming interface by burdening the programmers with the details of annotating memory operations to specify which operations must execute in program order.

Modern microprocessors employ aggressive instruction execution mechanisms to extract larger levels of instruction level parallelism (ILP) and reduce program execution time. To maximize ILP, these mechanisms allow instructions to execute both speculatively and out of program order. The ILP mechanisms buffer the speculative state of such instructions to maintain sequential semantics upon a misspeculation or an exception. The ILP mechanisms have reopened the debate about the memory models because they enable SC implementations to relax speculatively the memory order and yet appear to execute memory operations atomically and in program order [5,14,7].

An aggressive SC implementation can speculatively perform all memory operations in a processor cache. Such an implementation rolls back to the "sequentially-consistent" memory state if another processor is about to observe that the model constraints are violated (e.g., a store by one processor to a memory block loaded speculatively out of order by another). In the absence of frequent rollbacks, an SC implementation can perform potentially as well as the best of relaxed models—Release Consistency (RC)—because it emulates an RC implementation's behavior in every other aspect.

Gharachorloo et al., [5] first made the observation that exploiting ILP mechanisms allows optimizing SC's performance. Their proposed techniques are implemented in HP PA-8000, Intel Pentium Pro, and MIPS R10000. Ranganathan et al., re-evaluated these techniques [13] and proposed further optimizations [14] but concluded that a significant gap between SC and RC implementations remains for some applications and identified some of the factors contributing to the difference. Hill [7], however, argues that with current trends towards larger levels of on-chip integration, sophisticated microarchitectural innovation, and larger caches, the performance gap between the memory models should eventually vanish.

This paper confirms Hill's conjecture by showing, for the first time, that an SC implementation can perform as well as an RC implementation if the hardware provides enough support for speculation. The key observation is that both SC and RC implementations rely on reordering and overlapping memory operations to achieve high performance. While RC implementations primarily use software guarantees to enforce program order only when necessary, SC implementations rely on hardware speculation to provide the guarantee. So long as hardware speculation enables SC implementations to relax all memory orders speculatively and "emulate" RC implementations, SC implementations can reach RC implementations' performance. Any shortcoming in the hardware support for speculation prevents SC implementations from reaching RC implementations' performance.

In this paper, we identify the fundamental architectural and application requirements enabling an SC implementation to perform as well as RC implementations:

- Full-fledged speculation: Hardware should allow both loads and stores to bypass each other speculatively to avoid stopping the instruction flow through the processor pipeline. Current techniques [14,5] allow only loads to bypass pending loads and stores speculatively; stores are not allowed to bypass other memory operations. We present novel mechanisms to allow both loads and stores to bypass each other speculatively and yet appear to execute memory operations in program order.

- Large speculative state: Hardware should provide large enough speculative state for both processor and memory to allow out-of-order operations to hide long remote latencies. Without studying the required size of speculative state for processor or memory, previous studies proposed extensions to the re-order buffer for speculative processor state [14], but did not provide any support for speculative memory state beyond conventional load/store queues. We quantify the required size of speculative state for processor and memory, and provide speculative state support for both processor and memory.

- Fast common case: Hardware support for speculation should not introduce overhead (e.g., associative searches) to the execution's critical path. Previous proposals detect memory order violation for speculative loads [5,14]. We present fast and efficient mechanisms to detect memory order violation for both speculative loads and stores without excessive deterioration of processor pipeline critical paths.

- Infrequent rollbacks: The application should inherently incur infrequent rollbacks of speculative execution. We argue that well-behaved applications—i.e., applications benefitting from parallel execution on multiprocessors—indeed will not incur frequent rollbacks.

In our performance evaluation, we assume aggressive remote caching mechanisms and a large repository for remote data as suggested in most recent proposals for DSMs [10,11,4]. Using simulation of shared-memory applications, we show that our SC implementation, called SC++, achieves an RC implementation's performance in all the six applications we studied.

In Section 2, we describe the current implementation optimizations for SC and RC. In Section 3, we describe SC++. We present a qualitative comparison of current SC and RC implementations, and SC++ in Section 4. In Section 6, we report experimental results of our simulations, and in Section 7, we draw some conclusions.

2 Current ILP Optimizations

A memory consistency model defines the programming interface for a shared-memory machine. Sequential consistency (SC) provides the most intuitive programming interface by requiring that all memory operations execute in program order. To relax SC's requirement on ordering memory operations and increase performance, researchers and system designers invented many relaxed memory models. Relaxed memory models allow memory operations to execute out of program order but require the programmer to annotate those memory operations that must execute in program order to result in correct execution.

Processor vendors vary with respect to the memory models they provide [1]. HP and MIPS both adopt SC as the primary programming interface. Others provide a variety of relaxed models varying in the extent to which they relax memory ordering. Intel processors use Processor Consistency (PC) which allows loads (to one block) following a store (to a different block) to execute out of program order. Sun SPARC processors provide Total Store Order (TSO) which only relaxes store followed by load order and enforces order among all other memory operations. Sun SPARC, DEC Alpha, IBM PowerPC, all provide RC, which is the most relaxed memory model. RC allows memory operations (to different addresses) to execute out of program order. All relaxed models include special synchronization operations to allow specific memory operations to execute atomically and in program order.

Conventional implementations of memory consistency models executed the memory operations according to the model's constraint. For instance, SC implementations would execute memory operations according to the program order and one at a time. Modern microprocessors, however, exploit high degrees of instruction level parallelism (ILP) through branch prediction, execute multiple instructions per cycle, use non-blocking caches to overlap multiple memory access latencies, and allow instructions to execute out of order. To implement precise exceptions and speculative execution in accordance with sequential semantics, modern microprocessors use an instruction reorder buffer [15] to rollback and restore the processor state on an exception or a misspeculation. Aggressive implementations of a memory model can employ all these ILP techniques, which enable memory operations to over-

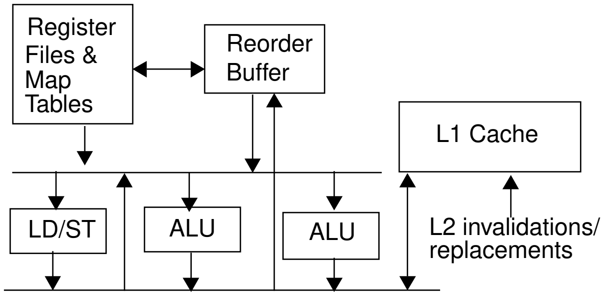

FIGURE 1: Speculative execution in current microprocessors.

lap and execute out of order but *appear* to comply with the memory model's constraints [14,5].

2.1 Mechanisms for Speculative Execution

In this section, we fist describe speculative instruction execution using ILP mechanisms in modern processors. We then present current memory model optimizations using these ILP mechanisms. We use the same pipeline model as Ranganathan et al., [13], which closely approximates the MIPS R10000 pipeline [17]. Figure 1 depicts the use of the reorder buffer (also referred to as an active window, or instruction window) to implement speculative execution and precise exceptions in modern microprocessors which issue instructions out of order.

The branch prediction and instruction fetch unit fetches and issues instructions. Upon issue, instructions are inserted in the reorder buffer. Upon availability of an instruction's operands, the instruction's (architectural) destination register is mapped to a physical register and is forwarded to a reservation station at each functional unit. The reorder buffer maintains the original program order and the register rename mapping for each instruction. Loads and stores are placed in the load/store queue, which acts as a reservation station but also maintains the program order among memory operations until the accesses are performed in the cache.

The pipeline forwards new register values generated by instructions to the reservation stations, and writes them to the reorder buffer and/or the physical registers. Instructions retire from the head of the reorder buffer in program order. Upon an exception or branch misprediction, all instruction entries in the reorder buffer following the mispredicted branch or the excepting instruction are rolled back and not allowed to retire from the reorder buffer [15]. Register rename-maps modified by the rolled back instructions are restored and execution is restarted at the offending instruction.

2.2 SC

In conventional SC implementations, the processor would faithfully implement SC's ordering constraints, performing memory operations atomically and in program order by issuing one memory operation at a time and blocking on cache misses. Such an implementation would be prohibitively slow in today's aggressive microprocessors because the processor must issue memory operations one at a time and the first cache miss would block both the cache and the instruction flow through the reorder buffer.

Gharachorloo et al., [5] proposed two ILP optimizations to improve shared memory's performance by preventing memory operations from frequently blocking the reorder buffer. Several current SC implementations (e.g., HP PA 8000, and MIPS R10000) include these optimizations. The idea is to use hardware prefetching and non-blocking caches to overlap fetching and placing cache blocks in the cache (or fetching block ownership requests) for the loads and stores that are waiting in the reorder buffer. Upon availability of the blocks in the cache, the loads and stores perform subsequently (and quickly) in the cache. Because the loads and stores retire atomically and in program order from the head of the reorder buffer, the prefetching optimization does not violate the memory model. Some implementations also retire pending stores from the reorder buffer but maintain program order in the load/store queue until they are performed.

Current aggressive SC implementations also allow loads to execute speculatively out of program order. Speculative execution allows loads to produce values that can be consumed by subsequent instructions while other memory operations (preceding the load in program order) are pending. The speculative load optimization is based on the key observation that as long as other processors in the system do not detect a speculatively loaded block, all memory operations appear to have executed atomically and in program order.

To guarantee the model's constraints, the speculative load optimization prevents other processors in the system from observing a speculative block. It is conservatively assumed that a speculatively loaded block may be exposed if it leaves processor caches—e.g., due to an invalidation message from or a writeback to the directory node in distributed shared memory (DSM). Therefore, the caches must hold a speculatively loaded block until the load retires. Upon a cache replacement signal from the L2 cache for a speculatively loaded block, however, the processor rolls back the load and all subsequent instructions (much as a branch misprediction) to restore the processor and memory to a "sequentially-consistent" state.

Because speculatively performed loads cannot retire from the reorder buffer until all pending memory operations are performed, a store at the head of the reorder buffer may block the instruction flow due to long remote latencies. But increasing the reorder buffer size to accommodate remote latencies may slow down processor critical paths involving associative searches through the buffer in a single cycle [12]. To alleviate this problem, speculative retirement [14] moves speculatively performed loads and subsequent instructions from the head of the reorder buffer to a separate history buffer before they retire. The history buffer maintains the information required to roll back, in case of an invalidation to a speculatively accessed block. Although speculative retirement narrows the performance gap between SC and RC implementations, a significant gap remains in some applications.

Store buffering [6] further enhances memory system performance by removing pending store instructions from the reorder buffer and placing them in the load/store queue. Relaxed models may realize the full benefits of store buffering by allowing loads in the reorder buffer to bypass pending stores. In conventional SC implementations, however, the reorder buffer stops retiring instructions at a load if there are pending stores and therefore, store buffering may not be as beneficial. Nevertheless, some commercial systems (e.g., HP processors) support store buffering for SC.

Both SC and RC implementations rely on reordering and overlapping memory operations to achieve high performance. The key difference between SC and RC implementations is that while RC implementations use software guarantees to guide the reordering and overlapping of memory operations, SC implementations use hardware speculation to reorder and overlap memory operations due to lack of any software guarantees. In spite of the above optimizations, SC implementations lag behind RC implementations because:

- the inability of stores to bypass other memory operations speculatively cause the load/store queue to fill up, eventually stopping instruction flow;
- long latency remote stores cause the relatively small reorder buffer (or the history buffer, in the case of speculative retirement) and load/store queue to fill up with speculative processor and memory state, respectively, stalling the pipeline;
- the capacity and conflict misses of small L2 caches cause replacements of speculatively loaded blocks, resulting in rollbacks.

2.3 RC

RC modifies the programming interface to allow the programmer to specify the ordering constraints among specific memory operations, so that in the absence of such constraints memory operations can overlap in any arbitrary order. Many microprocessors provide special fence instructions (e.g., the MEMBAR instruction in SPARC V9, or the MB and WMB instructions in Alpha) to enforce specific ordering of memory operations wherever needed. Typical RC implementations use special fence instructions at the lowest level to enforce memory ordering [6] but provide higher level programming abstractions for synchronization.

Conventional RC implementations achieved high performance primarily by using store buffering in the load/store queue to allow loads and stores to bypass pending stores and would maintain program order among memory operations only on executing a fence instruction. Modern RC implementations can additionally take advantage of hardware prefetching and non-blocking caches to fetch multiple cache blocks or make block ownership requests (for stores). Unlike SC implementations, RC implementations can use binding prefetches so that the loads can be performed before reaching the head of the reorder buffer. Moreover, RC implementations, like SC implementations, can also speculatively relax ordering across fence instructions and use rollback mechanisms if a memory model violation is detected by other processors.

3 SC++: SC Programmability with RC Performance

SC++, our implementation of SC, is based on the observation that SC implementations can approach RC implementations' performance if: (1) the hardware provides efficient mechanisms to relax order speculatively for not only loads, as done in [5], but also stores, (2) the system provides enough space to maintain not only speculative processor state, as proposed in [14], but also speculative memory state of reordered memory operations, (3) the support for speculation does not add excessive overhead to the processor pipeline, and (4) rollbacks are infrequent so that in the common case memory operations execute and complete with no ordering constraints, much as in RC implementations.

3.1 Speculative Execution in SC++

To fully emulate an RC implementation, SC++ relaxes all memory orders speculatively and allow instructions to continue to issue and execute at full speed even in the presence of pending long-latency store operations. To guarantee SC's constraints, SC++ maintains the state corresponding to all speculatively executed instructions between a pending store and subsequent (in-program-order) memory operations until the pending store completes. If there is an external coherence action (e.g., an invalidation of speculatively loaded data or external read of speculatively stored data) on speculatively accessed data, a misspeculation is flagged and execution is rolled back to the instruction that performed the speculative access. Thus, speculative state of loads and stores is not exposed to the other processors in the system, much as speculative loads are handled in [5].

Figure 2 illustrates SC++. SC++ supplements the reorder buffer with the Speculative History Queue (SHiQ) to maintain the speculative state for stores, much as speculative retirement does for loads. The SHiQ removes completed instructions as well as issued or ready to issue store instructions from the reorder buffer, allows instructions to retire and update the processor state and L1 cache speculatively, and maintains a precise log of the modifications to enable rolling back and restoring to the state conforming to SC's constraints. Thus, SC++ performs speculative stores to the cache itself instead of buffering the stores in the load/store queue, avoiding stalls caused by the filling up of the store queue due to long remote latencies. Upon completion of the earliest (in program order) pending store operation, the hardware disposes of all of the SHiQ's contents from the head until the next pending store operation. Since loads are moved to the SHiQ only after they complete in the reorder buffer, stores are the only operations in the SHiQ that may be pending; all other instructions in the SHiQ are (speculatively) completed instructions.

When an instruction retires from the reorder buffer, if there is a preceding pending store with respect to the instruction, the hardware inserts a modification log at the end of the SHiQ, recording the old architectural state that the instruction modifies. For instance, for an arithmetic instruction, the log maintains the physical register number, the old renaming map (i.e., the map prior to the instruction's execution), and the old value of the instruction's destination register.

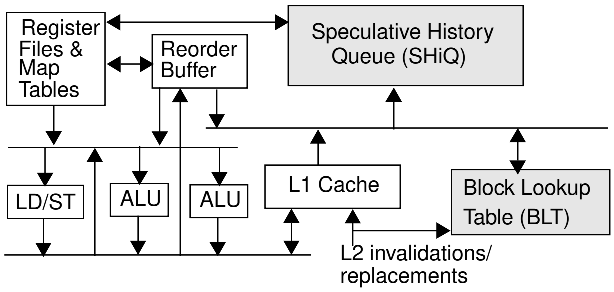

FIGURE 2: SC++ Hardware.

To speculatively retire store instructions while a preceding program-order store is pending, the hardware performs a read-modify-write cache access much as a cachable synchronization instruction (e.g., SWAP in SPARC) in modern microprocessors. Read-modify-writes, however, typically require an additional cycle to access the cache (e.g., Ross HyperSPARC). To prevent the slightly longer access latency of a read-modify-write operation from blocking access to the cache, the hardware can employ several well-known bandwidth optimizations to the L1 cache. Alternatively, by carefully scheduling speculative stores, the hardware can prioritize cache accesses to allow loads access the cache with a higher priority than speculative stores and thereby minimize the load-to-use latency among instructions.

3.2 Detecting Memory Order Violation

SC model requires the SC++ hardware to guarantee that relaxing the memory order is not observed by or exposed to the rest of the system. Our implementation (Figure 2) provides this guarantee by rolling back all execution state when a speculatively loaded or stored block is invalidated (by the DSM home directory), read (by a remote node, in the case of speculatively stored data), or replaced (due to capacity or conflict misses) from the lower-level L2 cache. In general, such an approach is conservative because SC++ need only to ensure that a speculative block does not leave a DSM node. Recent proposals for DSMs with aggressive remote caching techniques provide a large special-purpose remote access cache either in the network interface [10], or in both main memory and the network interface [11,4]. SC++ may limit the rollbacks to the less frequent case of speculative blocks leaving the remote cache.

Upon every invalidation, replacement or downgrade from L2, the hardware must determine whether the block has been accessed speculatively by a load or store. Because the SHiQ must be large enough to store the complete history of instruction execution during a pending remote memory operation, the queue may be too large to allow a fast associative search. Moreover, there may be frequent invalidations or replacements from L2 to blocks that are not speculatively accessed, necessitating a fast lookup.

To provide a fast lookup, SC++ uses a small associative buffer, called the Block Lookup Table (BLT), to hold a list of all the unique block addresses accessed by speculative loads and stores in the SHiQ. Unlike current SC implementations which identify speculatively loaded blocks by directly searching the reorder buffer and the load/store queues, the BLT decouples the search mechanisms to identify speculative blocks from the rollback mechanisms in the SHiQ that maintain all the speculative processor and memory state. The BLT is based on the key observation that loads and stores are only a fraction of all executed instructions and there is a high temporal and spatial locality in near-neighbor load and store instructions. As a result, a block lookup table can significantly reduce the search space as compared to the SHiQ.

3.3 Rolling Back Processor & Memory State

SC++ must roll back the processor and memory state to a “sequentially consistent” state upon a lookup match in the BLT. To guarantee forward progress and avoid livelocks/deadlocks, the hardware must restore all processor and memory state up to the first instruction in program order that speculatively accessed the matching block. Restoring the processor state involves stopping the pipeline and accessing the appropriate hardware structures. Restoring the speculatively stored data requires accesses to the local cache hierarchy, which may move the data from the lower levels to L1, if the speculative data is displaced from L1 to the lower levels. Because all of the data accessed by the instructions in the SHiQ are guaranteed to be present on the node, restoring the data can proceed without involving the coherence protocol.

Upon restoring the processor and memory state, the hardware inhibits further speculative retirement of instructions into the SHiQ until all pending stores have been performed. Such a policy guarantees forward progress by allowing the instruction causing the rollback to execute and retire (non-speculatively) in program order. During rollback, the processor also inhibits further coherence message processing to avoid deadlocks.

Depending on the rollback frequency and the desired performance in the presence of frequent rollbacks, the implementation can optimize the rollback process. A slow rollback will slow down both the faulting processor and any processors sending coherence messages to the faulting processor. One way to accelerate the rollback process is to exploit the processor ILP mechanisms to roll back multiple instructions per cycle. Another optimization includes allowing invalidation messages for read-only blocks to be immediately serviced eliminating the rollback waiting time for the response message. For blocks with speculatively stored data, a further optimization to eliminate the waiting time includes restoring the requested block first before the rollback process starts.

4 Qualitative Analysis

The primary difference between RC implementations and SC++ is that RC implementations rely on software to enforce the memory order necessary to guarantee correctness, whereas SC++ relies on hardware to provide such a guarantee. While RC changes the program interface to relax memory order, SC++ employs speculative mechanisms in hardware. In this section, we identify the application and system characteristics that enable SC++ to reach RC implementations’ performance.

To relax memory orders fully, SC++ must provide enough space to maintain the processor and memory state

	Relaxing Orders	Mechanisms to Guarantee Order	Potential for Order Violation
SC	loads bypass loads and stores	speculative execution using reorder buffer, load/store queue, and speculative placement of data in cache	lower
RC	loads and stores bypass each other between fences, loads bypass loads and stores across fences	fence instruction, speculative execution as in SC across fences	lower
SC++	loads and stores bypass each other	speculative execution using reorder buffer, load/store queue, and speculative placement of instructions in SHiQ, data addresses in BLT and data in cache	higher

Table 1: Comparison of implementations.

corresponding to all (out-of-program-order) speculatively executed instructions while a memory operation is pending. The state includes the processor cache hierarchy (and the remote cache) maintaining the speculatively accessed remote blocks, and the special-purpose buffers (e.g., SHiQ and BLT) maintaining the modification logs for the speculatively executed instructions. SC++ must also provide a fast mechanism to detect rollbacks because there may be frequent remote block replacements or invalidation messages in a communication-intensive application even though rollbacks are infrequent because processors tend to access different memory blocks at a given time.

Given all the speculative state, the only impediment for SC++ to achieve RC implementations' performance is the fraction of execution time lost to rollbacks. Unfortunately, the rollback penalty in SC++ may be rather high, because long latencies of memory operations create potential for a large number of speculatively executed instructions. However, we argue that rollback frequency in well-behaved applications is negligible.

A rollback occurs because two or more processors simultaneously access the same shared-memory blocks. There are three scenarios in which rollback frequency can be high: (1) there are true data races in the application, (2) there is a significant amount of false sharing, and (3) inevitable cache conflicts. Applications for which a significant fraction of execution time is spent accessing such data typically do not benefit from parallel execution in DSM because the overhead of communicating memory blocks across the processors dominates an execution time.

Table 1 compares the extent to which the memory model implementations relax memory order. Current aggressive SC implementations only relax memory order with respect to loads and use existing architectural mechanisms to execute instructions speculatively. RC implementations primarily relax order by requiring the software to guarantee correct placement of fence instructions. SC++ uses extra hardware to relax all orders speculatively and fully emulate RC implementations.

5 Experimental Methodology

Table 2 presents the shared-memory applications we use in this study and the corresponding input parameters. *Em3d* is a shared-memory implementation of the Split-C benchmark. *Lu* (the non-contiguous version), *radix*, *raytrace, water* (the nsquared version) are from the SPLASH-2 benchmark suite. *Unstructured* is a shared-memory implementation of a computational fluid dynamics computation using an unstructured mesh.

Application	Input Parameters
em3d	8192 nodes, 20% remote
lu	256 by 256 matrix, block 8
radix	512K keys
raytrace	teapot
unstructured	mesh 2K
water	343 molecules

Table 2: Applications and input parameters.

We use RSIM, a state-of-the-art DSM simulator developed at Rice university, to simulate an eight-node DSM. Every DSM node includes a MIPS R10000 like processor, first and second level caches, and main memory. Table 3 shows the base system configuration parameters used throughout the experiments unless otherwise specified. Our application data set sizes are selected to be small enough so as not to require prohibitive simulation cycles, while being large enough to maintain the intrinsic communication and computation characteristics of the parallel applications. Woo et al., show that for most of the SPLASH-2 applications, the data sets provided have a primary working set that fits in a 16-Kbyte cache [16]. Therefore, we assume 16-Kbyte (direct-mapped) processor caches to compensate for the small size of the data sets. We assume large L2 caches, as suggested by recent proposals for DSMs [10,4], to eliminate capacity and conflict misses, so that performance difference among the memory models is solely due to the intrinsics of the models.

Processor Parameters	
CPU	300MHz, 4-issue per cycle
reorder buffer	64 instructions
Load/store queue	64 instructions
L1 cache	16-Kbyte, direct-mapped
L2 cache	8-Mbyte, 2-way
L2 fill latency local	52 processor cycles
L2 fill latency remote	133 processor cycles
Cache line size	64 bytes

Table 3: Base system configuration.

In our experiments, all the memory model implementations use non-blocking caches, hardware prefetching for loads and stores, and speculative load execution. Neither the SC nor RC implementation uses speculative retirement (i.e., the history buffer). SC++ uses the SHiQ and BLT. Rollbacks due to instructions in the reorder buffer take one cycle to restart execution at the offending instruction. Any rollback due to instructions in the SHiQ (for SC++) is performed at the same rate as instruction retirement (i.e., 4 instructions per cycle).

6 Results

We start with a performance comparison of an SC implementation, an RC implementation, and SC++ in Section 6.1, which is the main result of this paper. We show that with unlimited SHiQ, SC++ does reach the RC implementation's performance; SC++ performs as well as the RC implementation even after limiting the SHiQ to a finite size. Section 6.2 presents results on the impact of network latency on the relative performance of the systems. Our results indicate that with larger network latencies, SC++ still keeps up with the RC implementation, albeit using larger speculative state, even though the gap between the SC and RC implementations grows.

We show that to close the performance gap, SC++ must closely emulate the RC implementation by overlapping all memory operations that the RC implementation overlaps and requiring the entire set of SC++ hardware—a large SHiQ with the associated BLT and a large cache. Future processor designs may have large reorder buffers, obviating the need for the SHiQ and BLT. Section 6.3 presents results indicating that increasing the reorder buffer size narrows the gap between the SC and RC implementations for many applications; the rest of the applications still require SC++ hardware to close the gap.

Our results in Section 6.4 indicate that performing stores in strict program order causes SC++ to be considerably slower than the RC implementation, confirming the need to execute stores speculatively. Finally, in Section 6.5, we show that with smaller L2 caches, rollbacks due to replacements of speculatively accessed blocks artificially widen the gap between the SC and RC implementations.

6.1 Base System

In Figure 3, we show the speedups of the RC implementation, SC++ using an infinitely large SHiQ (shown as SC++inf), and SC++ using a 512-entry SHiQ and a 64-entry BLT (shown as SC++S512B64) measured against the base case of the SC implementation. Although both the SC and RC implementations are equipped with non-blocking caches, prefetching, and speculative loads, there is a significant gap between the SC and RC implementations. On the average, the RC implementation is 18% better than the SC implementation, and at most, the RC implementation is 38% better than the SC implementation. The main reason for this gap is that, unlike the RC implementation, the SC implementation cannot retire any memory operations past a pending store. The gap is large in the case of *radix* because store addresses depend on previous loads, which stops the memory unit from issuing prefetches, leading to pipeline stalls for as long as the entire store latency. In the rest of the applications, the gap is less because both the SC and RC implementations stall for loads, making stores less important.

SC++inf performs as well as the RC implementation. By allowing stores to bypass other memory operations, SC++ closely emulates the RC implementation, closing

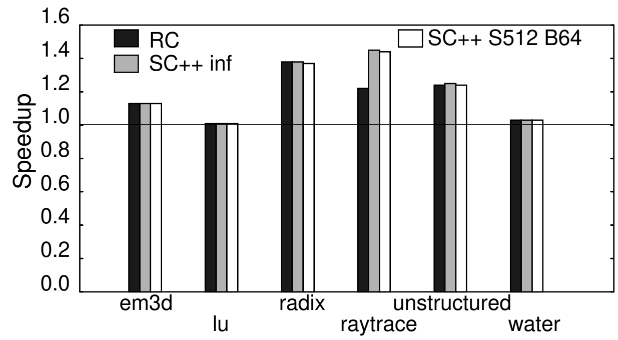

FIGURE 3: Comparison of SC, RC, and SC++.

This figure compares the speedups of the RC implementation and SC++ normalized to that of the SC implementation. SC++inf corresponds to an infinitely large SHiQ and BLT. The SC++ S512B64 corresponds to SC++ with a SHiQ of 512 instructions and BLT of 64 entries.

the performance gap. For all the applications, the number of memory order violations due to speculation is too small to have any effect on overall performance.

For all the applications, SC++S512B64 realizes the full benefits of SC++ with an infinitely large SHiQ. For *em3d*, *lu*, *water*, and *unstructured*, a SHiQ with fewer than 512 entries suffices. For *radix* and *raytrace*, 512 entries were needed to reach the performance of SC++inf. A BLT of size 64 was sufficient for all applications.

In the case of *raytrace*, SC++ performs better than the RC implementation by a wide margin. In this application, rollbacks in the SHiQ actually result in performance improvement! These rollbacks caused by looping reads of lock variable, prevent the injection of more messages into the network, reducing both network and lock contention. Kägi et al., showed that by simply using exponential backoff the performance of *raytrace* can be increased two-fold [8]. Although SC++ does not introduce exponential backoff, the time taken to restore the state on a rollback produces a similar effect.

6.2 Network Latency

In this section, we study the effect of longer network latency on the performance of the RC implementation and SC++. We increase the network latency to four times the remote latency of the base configuration described in Table 3. In Figure 4, we show the speedups of the RC implementation, SC++ using an infinitely large SHiQ (shown as SC++inf), SC++ using a 512-entry SHiQ and a 64-entry BLT (shown as SC++S512B64), and SC++ using a 8192-entry SHiQ and a 128-entry BLT (shown as SC++S8192B128) measured against the SC implementation. All the experiments use the longer network latency.

Compared to the performance gap between the SC and RC implementations shown in Figure 3, the gap in Figure 4 is larger for all the applications. On increasing the network latency by a factor of four, the gap increases from 18% to 31%, on the average. The RC implementation hides the longer network latency better than the SC implementation by overlapping more store latencies. For *em3d*, *raytrace*, and *unstructured,* the overall performance of the RC implementation (and the other implementations) decreases four-fold when compared to the faster network

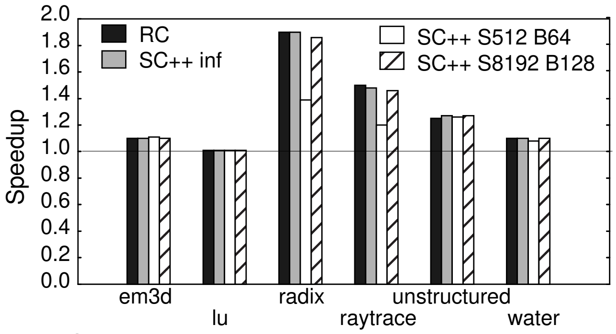

FIGURE 4: Impact of network latency.

The figure plots the speedups of the RC implementation and SC++ normalized to that of the SC implementation. The network latency was increased, for shown experiments, to eight times the local memory latency. Numbers following the letters 'S' and 'B', in the legend, correspond to the sizes of the SHiQ and BLT, respectively.

used in Section 6.1; for *lu*, *radix*, and *water* the decrease in performance is only by a factor of two, indicating that these three applications are less sensitive to remote latency.

In spite of the longer network latency, SC++inf keeps up with the RC implementation, showing that SC++ can closely emulate the RC implementation, achieving similar overlap of memory operations. Not surprisingly, the longer network latency creates a performance gap between SC++S512B64 and the RC implementation for *radix* and *raytrace*, indicating that a 512-entry SHiQ is insufficient to absorb the extra latency of remote memory operations. By increasing the SHiQ size to 8192 entries and the BLT to 128 entries, SC++ can perform as well as the RC implementation for *radix* and *raytrace*. For the rest of the applications, the smaller SHiQ and BLT configuration of SC++ performs as well as the RC implementation. Note that in the case of *raytrace*, even SC++S8192B128 no longer performs better than the RC implementation because the longer network latency dominates the lock acquisition patterns.

6.3 Reorder Buffer Size

To determine whether large reorder buffer sizes in future ILP processors will obviate the SHiQ and BLT, we study the effect of increasing the reorder buffer size on the performance of the SC and RC implementations. In Figure 5, we show the speedups of the SC and RC implementations at reorder buffer sizes of 64 and 1024 instructions, using the SC implementation with a 64-instruction reorder buffer as the base case. Note that although both the SC and RC implementations use non-blocking caches, hardware prefetching, and speculative loads, the SC implementation cannot retire stores out-of-order but the RC implementation can.

With a 64-instruction reorder buffer, there is a significant performance gap between the SC and RC implementations, as already mentioned in Section 6.1. Increasing the reorder buffer size to 1024 instructions, the gap shrinks for all the applications, except for *raytrace* and *unstructured*. Increasing the reorder buffer size from 64 to 1024 instructions shrinks the gap from 18% to 14%, on the average.

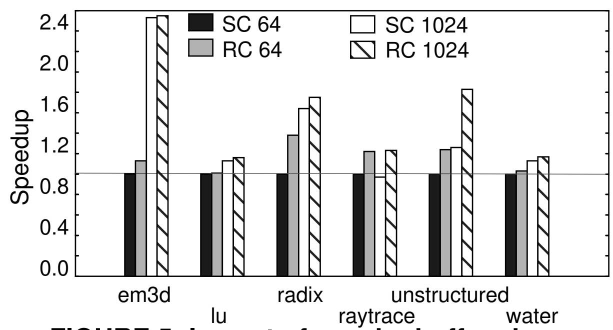

FIGURE 5: Impact of reorder buffer size.

The figure compares the speedups of the RC and SC implementations, for 64 and 1024 entry reorder buffer sizes, normalized with respect to that of the SC implementation with a 64-entry reorder buffer.

By hiding more store latencies through allowing more time for prefetches in a larger reorder buffer, the SC implementation performs closer to the RC implementation, although the RC implementation's performance improves as well. Although the gap between the SC and RC implementations shrinks on increasing the reorder buffer size, there is still a significant difference in performance between the two, suggesting that the SC++ hardware—the SHiQ and BLT—may be required to close the gap completely.

In the case of *raytrace*, increasing the reorder buffer size helps neither the SC nor RC implementation. A reorder buffer of 64 instructions already exposes the critical path through *raytrace,* so that larger reorder buffer sizes do not result in more overlap of memory operations. Performance of *raytrace* is mostly determined by the time spent in the critical sections of the program. Both the SC and RC implementations overlap the instructions in the critical section to the point where performance is limited by contention for the lock. The RC implementation's performance is better than that of the SC implementation because the RC implementation executes the critical section faster than the SC implementation. The RC implementation retires the stores in the critical section at a faster rate than the SC implementation, while the SC implementation incurs higher traffic due to more rollbacks. When the reorder buffer size is increased from 64 to 1024 instructions, the total number of loads issued per processor increases by 50% in the SC implementation, increasing the traffic significantly.

In the case of *unstructured*, the gap between the SC and RC implementations grows on increasing the reorder buffer size because the number of rollbacks in the case of SC increases. When the reorder buffer size is increased from 64 to 1024 instructions, the number of rollbacks increase by a factor of 35. These rollbacks increase the traffic in the case of the SC implementation, leading to a wider gap between the SC and RC implementations.

6.4 SHiQ Size and Speculative Stores

In this section, we show the importance of a large SHiQ and speculative stores to enable the SC implementation to reach the RC implementation's performance. In Figure 6, we show the speedups of the RC implementation, SC++

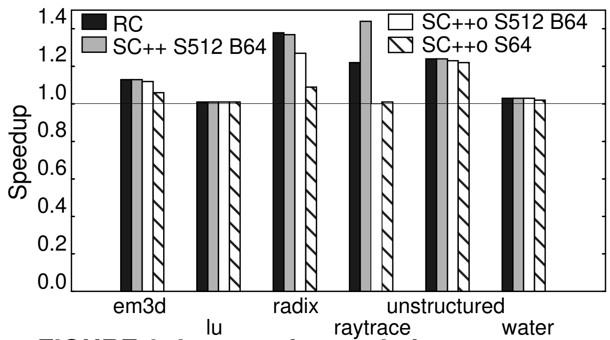

FIGURE 6: Impact of speculative stores.

The figure compares the speedups of the RC implementation, SC++ and SC++ without speculative stores (SC++o), normalized with respect to that of the SC implementation.

using a 512-entry SHiQ and a 64-entry BLT (shown as SC++S512B64), SC++ using a 512-entry SHiQ and a 64-entry BLT without speculative stores (shown as SC++oS512B64), and SC++ using a 64-entry SHiQ without speculative stores (shown as SC++oS64) measured against the base case of the SC implementation. The RC implementation and SC++S512B64 were compared in Section 6.1 and are shown here for reference.

Now, we compare SC++S512B64 with SC++oS512B64, which isolates the importance of speculative stores. SC++o can reach the RC implementation's performance for *em3d*, *lu*, *unstructured*, and *water*, which are not store-intensive. But for the cases of *radix* and *raytrace*, there is a significant gap of 9% and 22%, respectively, between the RC implementation and SC++oS512B64 because of their store-intensive nature. In these two applications, the absence of speculative stores causes significant performance loss. Not overlapping stores with other memory operations in SC++o leads to the filling up of the load/store queue which, in turn, blocks instruction issue, exposing the pipeline to remote latencies.

Reducing the SHiQ size from 512 to 64 entries in SC++o causes significant performance degradation for *em3d* and *radix*. The smaller SHiQ size significantly reduces the overlap among (non-speculative) stores and speculative loads, which exposes the pipeline to remote latencies. In the cases of *em3d* and *radix*, performance of SC++oS512B64 is 7% and 16%, respectively, better than that of SC++oS64.

6.5 L2 Cache Size

So far, we have compared the different implementations using large L2 caches for our simulations to avoid any capacity and conflict misses, so that performance differences among the memory models are solely due to the intrinsic behavior of the models. In this section, we show the importance of an L2 cache being large enough to hold all the speculative state of the SC implementation, in order for the SC implementation to reach the RC implementation's performance. In Figure 7, we show the speedups of the RC implementation and SC++ using a 512-entry SHiQ and a 64-entry BLT (shown as SC++S512B64) measured

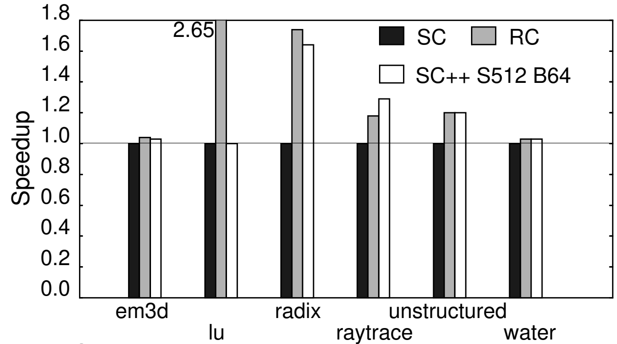

FIGURE 7: Impact of the L2 cache size.

The figure shows the impact of cache size on the SC implementation, RC implementation and SC++ performance. The L2 cache was reduced to 4-way 64-Kbyte size for shown experiments. The results were normalized with respect to the SC implementation.

against the base case of the SC implementation, using a 64-Kbyte, 4-way associative L2 cache.

There are two effects of a smaller L2 cache on the performance gap between the SC and RC implementations. On one hand, the gap may widen because the cache is not large enough to hold all of the SC implementation's speculative state. On the other hand, a smaller L2 cache may incur many load misses which slow down both the SC and RC implementations, resulting in a narrower performance gap between the two. For all the applications, except *lu* and *radix*, the higher load miss rate of the 64-Kbyte L2 cache degrades performance of both the SC and RC implementations, reducing the significance of the differences between the memory ordering constraints of SC and RC. Compared to the performance gap between the SC and RC implementations using the 8-Mbyte L2 cache (Figure 3), the gap between the SC and RC implementations using the 64-Kbyte L2 cache is wider for *radix* because conflicts on stores exposes remote latencies in the SC implementation.

In the case of *lu*, the striking gap between the SC and RC implementations using the 64-Kbyte L2 cache is primarily caused by rollbacks due to replacements (due to conflict misses in the cache) of speculatively accessed blocks. The number of rollbacks due to replacements increases inordinately (by a factor of 55,000), comparing the 64-Kbyte L2 cache with the 8-Mbyte L2 cache. For both *lu* and *radix,* although SC++ performs closer to the RC implementation than the SC implementation, SC++ is also sensitive to the rollbacks due to replacements.

7 Conclusions

This paper shows, for the first time, that SC implementations can perform as well as RC implementations if the hardware provides enough support for speculation. Both SC and RC implementations rely on reordering and overlapping memory operations to achieve high performance. The key difference is that while RC implementations primarily use software guarantees to enforce memory model constraints, SC implementations rely on full hardware speculation to provide the guarantee. Full-fledged hardware speculation can enable SC implementations to relax speculatively all memory orders and "emulate" RC implementations, enabling SC implementations to reach RC implementations' performance.

The fundamental architectural and application requirements that enable an SC implementation to perform as well as an RC implementation are: (1) hardware should allow both loads and stores to bypass each other speculatively to hide long remote latencies, (2) hardware should provide large speculative state, for both processor and memory, to allow out-of-order memory operations, (3) support for hardware speculation should not add excessive overhead to processor pipeline critical paths, and (4) rollbacks of speculative execution should be infrequent, as is the case for well-behaved applications.

Employing novel microarchitectural mechanisms, SC++ alleviates the shortcomings of current SC implementations to completely close the performance gap between SC and RC implementations. SC++ allows speculative bypassing of both loads and stores, yet appears to execute memory operations atomically and in program order. SC++ provides ample speculative state for the processor in the Speculative History Queue (SHiQ), which supplements the reorder buffer, to absorb remote access latencies. SC++ ensures sufficient speculative state for memory by placing speculative data in the local cache hierarchy itself and using a large L2 cache, as suggested by recent proposals for DSMs with aggressive remote caching techniques. SC++ uses the Block Lookup Table (BLT) to allow fast lookups of pending speculative accesses in the SHiQ, on an invalidation, downgrades or a replacement from the L2 cache. The SHiQ and BLT help minimize additional overheads to the processor pipeline critical paths.

Our experimental results obtained by software simulation show that SC++ achieves an RC implementation's performance in all the six applications we studied. Even at longer network latencies, SC++ can keep up with the RC implementation, albeit using larger speculative state. For SC++ to reach the RC implementation's performance, all the hardware of SC++—a large SHiQ with the associated BLT and a large cache—is needed. Simply increasing the reorder buffer size, without using the SHiQ or BLT, narrows the gap between the SC and RC implementations, but the extra mechanisms of SC++ are required to close the gap completely. Performing stores in program order causes SC++ to be considerably slower than the RC implementation, confirming the need to execute stores speculatively. Finally, smaller L2 caches cause rollback due to replacements of speculative blocks, artificially widening the gap between the SC and RC implementations.

8 Acknowledgements

We would like to thank Sarita Adve, Mark Hill, Alain Kägi, Vijay Pai, and the anonymous referees for their valuable comments on earlier drafts of this paper.

9 References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] Sarita V. Adve and Mark D. Hill. Weak Ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[3] M. Dubois, S. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[4] Babak Falsafi and David A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[5] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. I Architecture)*, pages I–355–364, August 1991.

[6] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[7] Mark D. Hill. Multiprocessors should support simple memory consistency models. 31(8), August 1998.

[8] Alain Kägi, Nagi Aboulenein, Douglas C. Burger, and James R. Goodman. Techniques for reducing overheads of shared-memory multiprocessing. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 11–20, July 1995.

[9] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[10] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[11] Adrian Moga and Michel Dubois. The effectiveness of SRAM network caches in clustered DSMs. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 103–112, February 1998.

[12] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[13] Parthasarathy Ranganathan, Vijay S. Pai, Hazim Abdel-Shafi, and Sarita V. Adve. The interaction of software prefetching with ILP processors in shared-memory systems. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 144–156, June 1997.

[14] Parthasarthy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1997.

[15] J. E. Smith and A. R. Plezkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, C-37(5):562–573, May 1988.

[16] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.

[17] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), April 1996.